Faculty of Computer & Information Sciences
Ain Shams University
Subject: DBA 271 File Organization
Year: (2nd year)undergraduate
Academic year: 2nd term 2019-2020

# Research Topic (9)

# Title: Indexed Allocation

**تحذير هام: علي الطالب عدم كتابة اسمه أو كتابة اي شيء يدل علي شخصيته**

## 1. Techniques for Block Allocation:

Why we need Techniques to allocate space on the Disk for Blocks?

For effective disk space utilization and, quick access as well as, reclaim the defragmented space on a disk as files Blocks are allocated and freed.

The major methods of allocating Blocks on disk space are:

A. Contiguous Allocation

B. Linked Allocation

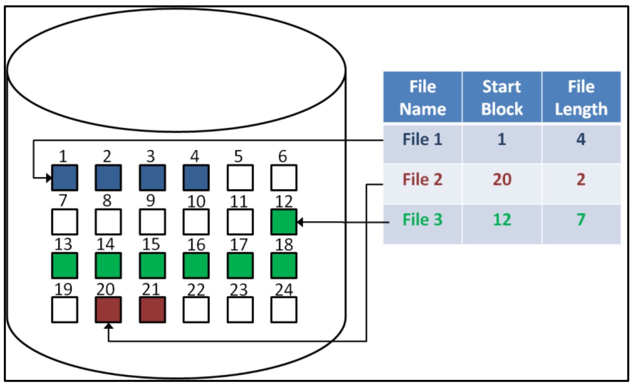
C. Indexed Allocation

## 2. Block Allocation Techniques Explained:

A. Contiguous Allocation:

In contiguous allocation, files are assigned to contiguous area of secondary storage (areas next to each other). This approach has the advantage that successive logical records are physically adjacent and require no head movement. Thus disk seek time is minimal, and record access speeds up.

Contiguous allocation keeps only the disk address (start of file) of the first block and the number of blocks allocated to the file.



| File Name | Start Block | File Length |
|---|---|---|
| File 1 | 1 | 4 |
| File 2 | 20 | 2 |
| File 3 | 12 | 7 |

Fig1. Contiguous Block Allocation

Advantages:-

- support both sequential and direct access:

The main advantage with this approach is that it can support both sequential and direct access with good performance because the location of data is previously known.

Disadvantages:-

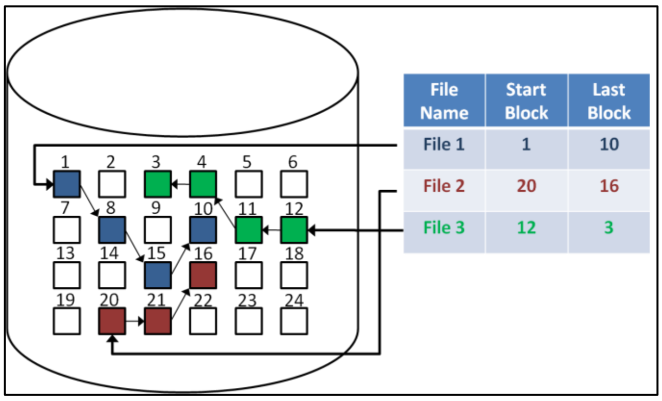
- External fragmentation:

The problem with this approach is that the allocation and de-allocation could result in regions of free disk space broken into chunks (pieces) within active space, which is called external fragmentation. External fragmentation occurs due to the existence of small free parts of disk space not enough to hold a file.

-Adding a new file could fail because of not finding contiguous space to store the file.

B. Linked Allocation:

Linked allocation is considered a disk-based version of the linked list. The blocks of disks can be distributed everywhere on the disk. The directory contains a pointer to the first and last block of the file (Analogy to Head and Tail in Linked List). Each block also contains a pointer to the next block.



| File Name | Start Block | Last Block |
|---|---|---|
| File 1 | 1 | 10 |
| File 2 | 20 | 16 |
| File 3 | 12 | 3 |

Fig 2. Linked Block Allocation

Advantages:

- Effective for sequential access
- It does not cause any external fragmentation since blocks can be placed anywhere.
- It also has the advantage that the file size need not to be known in advance and the file can grow freely.

Disadvantages:

- Since the blocks are not contiguous, this organization could lead to a lot of seeks (Performance affected badly).
- There is extra storage space required for the pointers.
- This scheme is not suitable for direct access since the location of a certain data item is not known in advance, but to find it you have to search the whole list of blocks.

C. Indexed Allocation

Index allocation addresses many of the problems of contiguous and Linked allocation. in this case, there is a separate one-level index for each file in the file allocation table; the index has one entry for each portion allocated to the file.

The file indexes are typically not stored physically as part of the file allocation table. Rather, the indexes for a file is kept in a separate block, and entry for the file in the allocation table points to that block.

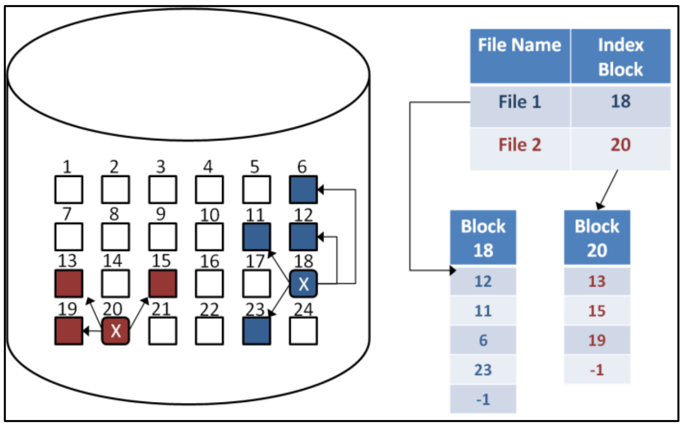


Fig 3. Indexed Block Allocation

The Advantages:

• It supports both sequential and random access )computed offset(.

• The search may be performed on the index blocks and these blocks can be kept close to each other to minimize seek time.

The Disadvantage:

- Internal fragmentation:

One of the problems with this organization is that the index block might not be used entirely, leaving some empty space inside this block (internal fragmentation).

-Also, the size of the file is limited by the size of the index block. Larger files would require either larger “index blocks” or using more than one index block.

Let’s focus on Indexed Allocation and how we solve larger files problem,

Single index block may not be able to hold all the pointers for files which are very large.
Following mechanisms can be used to use more than one index block:

- ➢ Linked scheme: This scheme links two or more index blocks together for holding the pointers. Each block of indexes would then contain a pointer or address to the next block.

- ➢ Multilevel index: In this policy, a first level index block is used to point to the second level index blocks which In turn points to the disk blocks occupied by the file. This can be extended to 3 or more levels depending on the maximum file size.

  Important: we will discuss B-trees in this Report, which can be used as dynamic multilevel indexes to guide the search for records in a data file.

- ➢ Combined Scheme (i-node scheme) used in UNIX: a combination of the previous two techniques which is used in the B+Trees. In this scheme.

3. B-Trees:

B-Trees are an extension of indexed allocation.

With keeping indexes on secondary storage and grow too large However, there are two fundamental problems associated with keeping an index on secondary storage:

• The index search has to be faster than the binary search.

• The insertion and deletion must be as fast as the search.

Trees like Binary Search Trees, AVL trees, appear to be a good general solution to indexing, but each particular solution presents some problems.

Multi-Level Indexing is a Better Approach to Tree Indexes

-Multiple keys are put into an index record, so higher level index refers to a lower level index, which means we build indexes of indexes.

And this technique "multi-record multi-level indexes" really help reduce the number of disk accesses and their overhead space costs are minimal, but inserting a new key or deleting an old one is very costly!

So, trees suffer from the fact that they are built downward from the top and that a "bad" root may unbalance the construct then Searching, insertion, and deletion becomes worst,
and Multilevel indexing takes a different approach that solves many problems but creates costly insertion and deletion.

An ideal solution would be one that combines the benefits of the previous solutions and does not suffer from the disadvantages they have.

B-Trees appear to do just that! B-Trees are multi-level indexes that solve the problem of linear cost of deletion and insertion.
It is used extensively now in indexing.

## 4. B-Trees Definition:

A B-tree is a self-balancing tree data structure that holds sorted data and allows logarithmic time searches, sequential access, insertions, and deletions. The B-tree generalizes the binary search tree, allowing nodes that have more than two children.

Unlike other binary search trees self-balancing, the B-tree is well suited for storage systems that read and write relatively large blocks of data, such as disks. It is commonly used in databases and file systems Why? B-Trees reduce the number of disk accesses which is very high compared to main memory access time.

B-Trees are built upward from the bottom rather than downward from the top, with B-Trees, we allow the root to emerge rather than set it up and then find ways to change it.

Formal Definition of B-Tree Properties:

1) All leaves are at same level (every path from the root to a leaf has the same length).

2) The B-Tree is defined by the term minimum degree ‘t’. The value t is dependent on the size of the disk block.

3) Each node except root must have at least t-1 keys. . Root may contain minimum 1 key.

4) All nodes (including root) can contain up to 2 t – 1 keys at most.

5) The number of children in a node is equal to the number of keys it contains plus 1.

6) All node keys are sorted in increasing order. Between two keys k1 and k2 the child contains all keys within the range from k1 to k2.

7) B-Tree grows and shrinks from the root which is unlike Binary Search Tree as mentioned.

8) Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(Log n).

9) The leaf level forms a complete, ordered index of the associated data file.

10) Every node (except the root) is at least half full.

## 4. How B-trees Work:

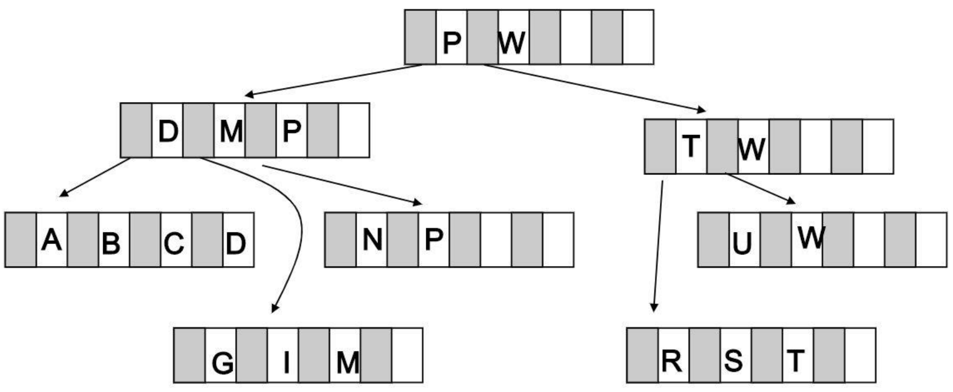
B-tree uses an collection of entries for a single node and for each of these entries it have a reference to child node.

Every node inside the B-tree is of the form

<P1, <K1, Pr1>, P2, <K2, Pr2>, ..., <Kq−1, Prq−1>, Pq>

where q ≤ p.
(Pi) a tree pointer, a pointer to a different node in the B-tree.
(Pri) is a data pointer to a record whose search key field value is equal to Ki (or the data file block that contains that record).

Within each node, K1 < K2 < ... < Kq−1 (increasing Order of Keys)



**Note:** references to actual record only occur in the leaf nodes.The interior nodes are only higher level indexes (this is why there are duplications in the tree)

Every B-Tree node is an Index Record. Each of these records contains the same maximum number of key-reference pairs called the B-Tree order. There is also a minimum number of key-reference pairs in the records, typically half the order.

Insertion and deletion from a B-tree:

B-tree has constraints that ensure that the tree is always balanced and that the space wasted by deletion, if any, never becomes excessive. The algorithms for insertion and deletion, though, become more complex in order to maintain these constraints. Inserting an item can cause splitting of a node. Deleting from a tree sometimes requires rebalancing of the tree.

For insertion, we first find the appropriate leaf node into which the inserted element falls (assuming it is not already in the tree). If there is already room in the node, the new element can be inserted simply. Otherwise the current leaf is already full and must be split into two leaves, one of which acquires the new element. The parent is then updated to contain a new key and child pointer. If the parent is already full, the process ripples upwards, eventually possibly reaching the root. If the root is split into two, then a new root is created with just two children, increasing the height of the tree by one.

Deletion works in the opposite way: the element is removed from the leaf. If the leaf becomes empty, a key is removed from the parent node. If that breaks invariant 3, the keys of the parent node and its immediate right (or left) sibling are reapportioned among them so that invariant 3 is satisfied. If this is not possible, the parent node can be combined with that sibling, removing a key another level up in the tree and possible causing a ripple all the way to the root. If the root has just two children, and they are combined, then the root is deleted and the new combined node becomes the root of the tree, reducing the height of the tree by one.

## 3. Advantages and Disadvantages of B-trees:

Let’s Summarize Advantages and Disadvantages of B-trees
Advantages of B-tree usage for storage systems that read and write relatively large blocks of data:

- keeps keys in sorted order for sequential traversing
- uses a hierarchical index to minimize the number of disk reads
- uses partially full blocks to speed insertions and deletions
- keeps the index balanced with a recursive algorithm
- A B-tree minimizes waste by making sure the interior nodes are at least half full. A B-tree can handle an arbitrary number of insertions and deletions.
- B-Trees can be stored in a file and B-Tree nodes can be read on an as needed basis allowing B-Trees to be larger than available memory.

In addition, its use in databases, the B-tree is also used in filesystems to allow quick random access to an arbitrary block in a particular file. The basic problem is turning the file block address into a disk block (or perhaps to a cylinder-head-sector) address.

what is the Disadvantages of B-trees? Well, The Question is not completed, the right one Disadvantages Compared to what? :

To B+tree?

B-tree has lower branching factor when compared to b+tree, which increases tree height and average number of disk seeks.

To Hash table?

A B-tree that doesn't fit RAM requires more disk seeks on-accurate average than a hash table, when performing a random lookup. B-tree is much more complicated than a hash table to implement.

To Array?

If the keys are dens, then the random access of an array is faster, and wastes less memory than b-tree. B-tree is much more complicated to implement than an array.

In RAM?

B-trees are not very efficient, when compared to other balanced trees, when they reside in RAM. Inserting or deleting elements involve moving many keys/values around. A b-tree has a chance to be efficient in RAM, if it has a very low branching factor, so that a node fits a cache line. This may lead to cache miss minimization.

## 6. Implementation:

In this section add a description of your implementation, you can add screen shots if needed.

## 7. Test Cases:

| Test Case | sizes of files inserted |
|---|---|
| 1 | 3+(1)+2+(1)+5+(1)=13 blocks=26KB |
| 2 | 8+(1)+1+(1)+4+(1)=16 blocks=32KB |
| 3 | 14+(1)+4(1)+2+(1)=23 blocks=46KB can not exceed 40KB , so almost file2 will rejected by the System |
| 4 | 6+(1)+6+(1)+5+(1)= |
| 5 | |

**Test Case 1:**

| **File** | **Index block** | **Data Blocks** |
|---|---|---|
| **1** | 16 | [15,1,20] |
| **2** | **11** | **[9,7]** |
| **3** | **14** | **[13,19,12,6,10]** |

**Test Case 2:**

| **File** | **Index block** | **Data Blocks** |
|---|---|---|
| **1** | 3 | [8,18,7,5,14,16,4,11] |
| **2** | **2** | **[19]** |

| | | |
|---|---|---|
| **3** | **13** | **[6,20,1,12]** |

**Test Case 3:**

| **File** | **Index block** | **Data Blocks** |
|---|---|---|
| **1** | 16 | [1,5,11,14,10,8,15,20,17,12,6,4,7,19] |
| **2** | **18 (Block id:18 it is actually empty i.e can be overwritten by next files).** | **[]** |
| **3** | 3 | **[13,2]** |

**Test Case 4:**

| **File** | **Index block** | **Data Blocks** |
|---|---|---|
| **1** | | |
| **2** | | |
| **3** | | |

**Test Case 5:**

| **File** | **Index block** | **Data Blocks** |
|---|---|---|
| **1** | | |
| **2** | | |
| **3** | | |

## 8. Conclusions

…………………………………………………………………………………...

…………………………………………………………………………………...

…………………………………………………………………………………...

## References:

[1] Folk, Michael J.; Zoellick, Bill (1992), File Structures (2nd ed.), Addison-Wesley, ISBN 0-201-55713-4

[2] B-tree,Wikipedia https://en.wikipedia.org/wiki/B-tree#Definition.

[3] Aho, Hopcroft, and Ullman, Data Structures and Algorithms, Chapter 11.

[4]Lectures Slides and Notes #5,#7

[5] http://www.brainkart.com/article/Dynamic-Multilevel-Indexes-Using-B-Trees-and-B--Trees_11526/

[at least 5 references]

*With My Best Regards,*

*your signature*